Homework 2
File Sharing and File Search

CS-438 - Decentralized Systems Engineering, Fall 2020

Publish date: Monday, October 26, 2020

Due date: Tuesday, November 17, 2020 @ 23:55

General Guidelines

In this homework, you will continue to build more functionalities to *Peerster* **on top of your Homework 1 implementation**. In this homework, you will add the functionalities to share files and search for files.

We will provide you with the skeleton files that contain the main interfaces to be implemented, test framework, and some useful functionality and implementation hints. The skeleton files are implemented on top of the skeleton for Homework 1, which means that you should be able to merge it with your existing implementation. Specifically, you will receive access to a new repository cs438-hw2-student-*name* which you can add as a new git remote in the folder with your existing implementation and pull the updates into your master branch, e.g., by running

```
git remote add hw2
https://gitlab.epfl.ch/cs438/students/cs438-hw2-student-XYZ.git

git pull --rebase hw2 master
```

Then you should be able to merge the master branch into the branch with your implementation. We suggest that you use an automatic tool, such as KDiff3, for merging (it can automatically resolve the conflicts of modifying the same file as long as the changes do not touch the same source lines), although you will still likely have to manually resolve some of the conflicts, for which you need use your own reasoning.

Note that you *must* use the new hw2 repository (`cs438-hw2-student-name`) to work on this homework, i.e. to make new commits and run tests.

Throughout this development and debugging process, you are welcome to discuss challenges and techniques with your fellow students, exchange pointers to relevant information or algorithms, debugging tips, etc., ***provided you each write your own code independently*****. Homeworks are individual per student.**

Teaching assistants will be available on Zoom every Friday 15:15-17:00, to discuss with you how to architect your implementation. ***TAs are not going to debug your code***, but they can help you ask the right questions just like your software engineer colleagues will do in the future. Room INF 1 is available every Monday from 13:15 to 15:00 for you to hack together and test your implementations, without TA supervision.

Third Life, a.k.a. Code Review

Every homework builds on the previous one and requires a full and working implementation of the previous homework. Thus, we strongly encourage you to fully implement every homework. However, if it so happens that: (1) You were unable to complete a homework assignment, or (2) You fully implemented your homework but the poor code design makes it hard for you to build on top, or (3) You are unsure why you don't pass all tests, we offer you an alternative.

After the deadline of Homework 1, you will receive 3 random submissions of your colleagues (by getting a read-only access to their HW1 repositories on Gitlab). You can choose to build on top of any of these 3 assignments for your next homework. **If you decide to do so, you need to specify that homework's identifier, i.e., the Gitlab repo link, in your future homework submission on Moodle.** Also, if you build on someone else's code, we *strongly suggest* that you review that colleague's code on Gitlab as a token of appreciation -- the review is, however, not graded.

However, be warned that there is no guarantee regarding the quality of these 3 randomly assigned submissions - they might be less good than your own implementation. Your best strategy, thus, is still to complete the homeworks yourself.

Introduction

We will enable Peerster nodes to share files, as well as to search for files shared by other nodes. For file sharing, each node will have the following functionalities: (1) Select the files it wants to share; (2) Announce the shared files to other nodes; and (3) Ask to and download a file from a specific peer. For file search, each node will additionally be able to (4) Enter a list of keywords and receive a list of matching files shared by other nodes; and (5) Download a file, possibly from several peers, but without specifying these peers.

Part 1: File Sharing

Peerster nodes will be able to send and receive potentially large files, not just short text messages. Because Peerster uses UDP for inter-node communication, which does not have congestion control, we have to break files up into chunks for transfer so as to not flood the network. To keep track of file chunks, Peerster uses hash trees (as many P2P systems do), which identify both complete files and parts of files. [1]

File Selection

To share files, you first need to enable the client to specify a file to share. In the next step, you will index shared files and divide them into chunks.

The GUI that we provide has a "Share File..." option (i.e., button), which opens a file selection dialog box that allows the user to select a file to share. If a user wants to share multiple files, they need to share them one at a time (only specific to the GUI). Due to the privacy protection mechanisms in the modern browsers that hide the access path to files, our GUI supports sharing only from the precreated `GUIShare/` folder (again it is only specific to the GUI and should not extend to the cli in your code). The GUI also displays the list of files already shared by the user.

You will need to implement file sharing for the CLI. For that, add a flag `-share` flag that specifies list of files to share, separated by comma, as follows:

```
./client -UIPort=8080
-share=<path1>/flyingDrone.gif,<path2>/report.txt

Usage of client:

   -  UIPort string
           port for the UI client (default “8080”)
   -  share string
           list  of  comma-separated  files  to  be  shared  by  the
      gossiper
```

File Paths

The gossiper stores the data/state about the shared files—whichever data you believe is necessary—in the directory whose name it should be possible to specify with the `-sharedir` flag (when starting a new gossiper). In the skeleton, you will find `_SharedData/` as the default value for this flag. In addition, you need to specify a name for the directory to store downloaded and reconstructed files using the `-downdir` flag. This flag takes `_Downloads/` as the default value. **Your implementation needs to automatically create the corresponding directories if they do not exist.**

Note that there should be no need to copy the files you share (e.g., from `path1` and `path2`) to this directory, although this is still possible and you might choose to, for example, store chunks there.

File Size

For simplicity, we assume that shared files should not be bigger than 2 MiB. The client should check for this condition and otherwise return an error with the message “Cannot share file, file named <name> exceeds 2 MiB”. The reason for this constraint will be more clear below.

File Indexing

The gossiper indexes as follows each file that the user shares via the GUI and CLI dialog above:

1. It divides the file into chunks and computes the hash of the contents of each chunk.
2. It computes a *MetaFile*, which is simply a file containing the hashes of each chunk.
3. It computes the hash of this MetaFile, the so-called *MetaHash*.

The MetaHash serves as the unique identifier for a file.

Chunks

To split a file into chunks, Peerster uses a **fixed chunk size of 8KiB (8192B)**. Most files are not a natural multiple of 8KiB in size; thus, the last chunk will likely be smaller than 8KiB in size. **Do NOT pad** the file's last chunk to 8KiB, because that would "lose" the file's correct original size.

MetaFile and MetaHash

To build up the **MetaFile corresponding to a shared file**, simply concatenate the 32-byte SHA-256 hash of each chunk. That is, store the hashes, in the order `chunk_0chunk_1...chunkN-1chunk_N` (no spaces, commas, etc) into one large `[32×N]byte` slice and write this slice to the binary MetaFile. Once you have this MetaFile, compute its SHA-256 hash (named `MetaHash`) and store it in your gossiper.

To compute SHA-256 hashes, please use the `sha256` hash function in the Go `crypto` package [2].

As mentioned above, `MetaHash` uniquely identifies a file. One consequence is that two files with identical contents but different names have the same `MetaHash`. Thus, copying a file under a different name, without changing the contents, and sharing it, should not result in creating a different MetaFile. The copy can simply use the same chunks, `MetaHash` and MetaFile as the original.

For simplicity, we impose the restriction that shared files are not bigger than 2 MiB, which means there's no need to chunk the MetaFile, as it is smaller than 8 KiB.

Gossiper data structures

After scanning a file to be shared, your Peerster should have the following metadata in its internal data structures, for each file:

- File name on the local machine, including the path.
- The MetaFile created as described above.
- The SHA-256 hash of the MetaFile.

File Download

Finally, we need a protocol for one node to retrieve a file from another node. In Part 1, we assume the retrieving node already has the `MetaHash` of the desired file and that it knows a peer that shares a file with that `MetaHash` (Part 2 of this homework addresses how the retrieving peer obtains this information). Peerster uses a simple one-chunk-at-a-time request / response download protocol.

Downloaded files are by default shared with other peers. This means, among others, that the downloaded chunks must be available to other peers even before the file is fully downloaded.

Overview

Any file download starts by first requesting the MetaFile. Thus, any peer who is downloading a file has the corresponding MetaFile, even when the file download did not complete yet (not all chunks have been downloaded).

For file download, nodes send data requests and replies for MetaFiles and chunks. Nodes route these messages “point-to-point”, similarly to routing private messages (Homework 1, Part 2), so that a node can download a file from another node that is connected either directly or only indirectly via intermediate hops. Similarly, requests and replies have `Destination` and `HopLimit` fields, which are the only fields the routing logic needs to care about. Thus, before doing any local processing, your code can decide simply by looking at these fields (`Destination` and `HopLimit`), whether to route a message on to another node or to process it at the local node. The `HopLimit` has a default initial value of 10. The routing logic is the same as for private messages: when a node receives a request or reply, the node processes the packet if it is the destination. Otherwise, if the `HopLimit` is zero, the node drops the packet, contrarily the node decrements the `HopLimit` and forwards the packet.

When the gossiper collects all the chunks for a given file, the gossiper reconstructs it and saves it locally in the download directory that is given to the gossiper as a parameter at start-up (as described above, the default value is `_Downloads)`. By default, all the files stored in the download directory are considered shared with the other peers. You will use the value specified with the `-filename` flag to name the file you downloaded. The name given with `-filename` does not need to be the same as the name of the file as shared by other peers, because it is impossible to know the name of the file on remote peers.

Details

The node requesting the file first sends a data request to download the MetaFile. The node waits for a reply to that request, retransmitting the request periodically if it does not receive a reply. Specifically, the node implements an exponential back-off mechanism: there are at most 5 retransmissions, each at a 2^i second interval, for i = [1, 5]. This means, the node retransmits after 2 seconds from the initial transmission if it does not receive a reply or if the received reply is invalid (see below), then again after 4 seconds etc. Then the node sends a request for each of the file's data chunks in turn, retransmitting the chunk request periodically just like above. This means that your gossiper needs to keep state for the files that are being downloaded. For the sake of simplicity, we ask you to **download chunks of a file one by one**, i.e. when your gossiper sends a request for a chunk, it should wait for the reply before sending the next request. In the real world, several chunks can be downloaded in parallel, however, for the purposes of this homework, the sequential approach suffices.

When a gossiper receives a request for a MetaFile or a chunk that the gossiper does not have, the gossiper simply drops the request. Such a request could be part of a malicious scenario or an honest mistake, and properly handling them is out of scope for this assignment. For those interested, a more efficient way - that doesn't lead to an avalanche of repeated requests from the sender - could be to issue a reply containing an empty `Data` field (the data structures are below). A reply with an empty `Data` field would not be considered invalid. However, you're not required to implement such a mechanism.

For a downloaded chunk / MetaFile to be valid, the SHA-256 hash of the chunk / MetaFile must match the hash (`HashValue` field) carried in the reply. Your code must check this invariant when it receives a MetaFile / chunk reply and drop the packet if the check fails.

Performance

As described above, you can keep the protocol simple by requesting only one chunk at a time for a given file download. You need, however, to support parallel file downloads on your gossiper, where **each download** deals with one chunk at a time. You also need to support parallel MetaFile downloads. You should make sure that your node's UDP buffers are large enough to handle incoming chunks.

Your protocol should allow that a peer, who has only a few chunks of a file (e.g., because it has only started downloading the file), can already offer those chunks to other peers.

You should carefully consider how to store the chunks of a downloaded file. Recall that downloaded files are by default shared. If a peer requests some specific chunk, your gossiper should not need to reparse the whole file.

We require that the state about shared files survives restarts, although the implementation of this persistent state does not need to be particularly efficient. For example, upon starting, the gossiper could automatically index all the data in its shared data folder.

Client

The GUI that we provide has three fields: (1) Destination node, (2) ***Hexadecimal*** `MetaHash`, (the `hex.EncodeToString()` and `hex.DecodeString()` are your friends here) and (3) The name that the downloaded file receives when saved locally. Note that the file name specified for local storage could but does not need to correspond to the file's name as stored on the destination node.

You need to add the same functionality to the CLI:

```
./client -UIPort=8080 -dest=anotherPeer -filename=flyingDrone.gif
-request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe699
2be5318
```

```
Usage of client:

   -  filename string
            name used to save the file on the local computer
   -  dest string
            peer identifier to download the file from
   -  request string
            the hexadecimal MetaHash value of the file to be
            downloaded
```

The only flag combinations valid for the `client` are the ones specified in the assignments, under client usage. ***Any other*** flag combination (e.g, `-request` and `-msg`, etc) should output an error on the client (`ERROR (Bad argument combination)`) and return error code 1 (`os.Exit(1)`). Also, if the `-request` flag has a wrong format, output an error `ERROR Unable to decode hex hash)` and return code 1.

Data structures

Add support for two new messages to your protocol: 1) For chunk and MetaFile requests; and 2) For chunk and MetaFile replies. `DataRequest` and `DataReply` are now options for encapsulation in a `GossipPacket`. The message formats are below:

- **DataRequest** message
  JSON format: `{Origin: <value>, Destination: <value>, HopLimit: <value>, HashValue: [byte]}`

- **DataReply** message
  JSON format: `{Origin: <value>, Destination: <value>, HopLimit: <value>, HashValue: [byte], Data: [byte]}`

`HashValue` represents:
- either the hash of the requested chunk
- or the `MetaHash`, if the request is for a MetaFile

`Data` is the actual data (MetaFile or a chunk) that a node replies to a request with.

Suggested standard output

For ease of debugging, we suggest the following output, which our reference also uses. Note, however, that the output format is merely a suggestion and is not required for passing the tests. Feel free to disable / change the output as you wish.

Output format for `./client -UIPort=8080 -dest=anotherPeer -filename=flyingDrone.txt -request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318` (assume the file has 3 chunks, which the gossiper can see from the MetaFile)

```
DOWNLOADING MetaFile of flyingDrone.gif from anotherPeer
DOWNLOADING flyingDrone.gif chunk 1 from anotherPeer
DOWNLOADING flyingDrone.gif chunk 2 from anotherPeer
DOWNLOADING flyingDrone.gif chunk 3 from anotherPeer
RECONSTRUCTED file flyingDrone.gif
```

Watchers

The testing framework uses the Watchers. Just as in Homework 1, you need to send all the incoming packets to `inWatcher` and all the outgoing packets from your gossiper to `outWatcher` via the `Notify` method of the watcher. Specifically, you need to wrap the gossip packet in the `CallbackPacket` struct that you will find in `gossip/packets.go`. You do not need to modify the code of the watcher, simply send in the incoming and outgoing gossip packets.

Part 2: File Search

Now that we can share and download files, we need a more convenient way to search for interesting files to download. Peerster nodes will support keyword-based search, where a file is a match if its name contains any of the keywords, **using as regular expression `*keyword*`, where `*` represents 0 or more characters**. Peerster implements the file search using a simple expanding-ring flooding scheme (the concept should be familiar from the lectures).

File search uses the notion of budget to limit the number of nodes that process the search request. When a node searches for a file, it sends a search request that contains the keywords and a search budget. The Peerster node R that receives a search request: (1) First processes the request locally, searching among the shared files (local and downloaded) for any file names matching any of the search keywords, and sending a search reply as described below if any files match. (2) Then, R subtracts 1 from the incoming request's budget. Only if the budget B is still greater than zero, R sends the request to up to B of R's neighbors, **excluding the node that R has just received the request from**. If R has more than B neighbors, R forwards the request to randomly chosen B neighbors. If R has fewer than B neighbors, it divides the remaining budget B as evenly as possible (i.e., plus-or-minus 1) among the recipient nodes of this search request.

For example, if an incoming search request has a budget of 3 and the receiving node has 5 neighbors, the node first processes the request locally, subtracts 1 for itself, then forwards the request to 2 other randomly-chosen neighbors, giving each forwarded request a budget of 1. Alternatively, if the incoming request's budget is 10, the node first subtracts 1, then forwards the request to all 5 neighbors, such that 4 forwarded requests have a budget of 2 and the remaining one has a budget of 1.

A search reply is a point-to-point message to the node that issued the search. The reply contains several search results, one for each local file that matches the search. Each search result contains the name of the file that matched the keyword search, the MetaHash of the file and the chunks that the replying peer has locally. The search reply contains a hop limit field, set to a default value of 10, and routed similarly to data requests and replies.

When the client performs a search but does not specify a budget, the gossiper should start with a search query budget of 2, then periodically, **once per second**, repeat the query, doubling the budget each time, until you reach the maximum budget of 32 (inclusive) or obtain a threshold number of files that are a total match. A total match means all the chunks of the matched file are present at at least one peer. To compute the number of chunks that a file has, the gossiper downloads the MetaFile. Recall that any peer that has a file chunk also has the MetaFile of the corresponding file, and one can compute the number of chunks of a file from its MetaFile. You can get the MetaFile of a file at random from any peer that has a chunk of that file.

Finally, the gossiper needs to be able to download one of the files that fully matched (the file is specified by the client using the MetaHash). The gossiper needs to keep state after the search command to remember which peers have which chunks. With this state, the gossiper should download all the chunks of the specified file **from peers (choosing a random peer for each chunk) among those that replied with a search reply specifying that they have file chunk(s) for that file**. Recall that reconstructed files should be stored in the download folder specified with the `-downdir` flag.

Details

Peers need to detect if they receive a duplicate search request and ignore it. Duplicate means that the origin and keywords in a search request are the same as in another search request received in the last 0.5 seconds.

You can assume that all search reply messages correspond to a previously issued search request, which means there are no unsolicited or malicious search reply messages.

Data structures

You will then need to add two new message types to the protocol to handle searches: `SearchRequest` and `SearchReply`, which are now options for encapsulation in a `GossipPacket`. The message formats are below:

- **SearchRequest** message
  JSON format: `{Origin: <value>, Budget: <value>, Keywords: [<value>]}`
- **SearchReply** message
  JSON format: `{Origin: <value>, Destination: <value>, HopLimit: <value>, Results: [*SearchResult]}`
    - **SearchResult**
      JSON format: `{FileName: <value>, MetaHash: [<value>], ChunkMap: [<value>]}`

Client

The provided GUI has an option to search for a file by keyword, by allowing the user to enter one or more keywords. It also has a field for entering the search budget (optional). The GUI accepts a line of text, and treats comma characters as keyword separators: `keyword1,keyword2,etc.` The GUI displays the file matches obtained in a list in the order the matches are received, so that matches received earlier show up at the top of the list. To let the controller know about the received matches, you need to make a copy of `GossipPacket` for each matched file with this file being the only element of `SearchResult` and send it to the client `NewMessage` callback.

You need to add to the CLI functionality for file search, as follows:

```
./client -UIPort=8080 -keywords=share,ex -budget=2

Usage of client:

   -  keywords string
            comma separated list of keywords
   -  budget uint64
            optional search budget; if missing, then the gossiper
            starts with a budget of 2 and increases it (see above)
```

You need to add to the CLI functionality for file download from random peers (described above):

```
./client -UIPort=8080 -filename=flyingDrone.gif
-request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe699
2be5318
```

```
Usage of client:

   -  filename string
            name used to save the file on the local computer
   -  request string
            download the file (MetaFile and all chunks) of the file
            with this hexadecimal MetaHash
```

Suggested standard output

For ease of debugging, we suggest the following output, which our reference also uses. Note, however, that the output format is merely a suggestion and is not required for passing the tests. Feel free to disable / change the output as you wish.

- When receiving a search reply from peer named <peer>, print at stdout for each matching search result:

```
FOUND match <filename> at <peer> MetaHash=<MetaHash> chunks=<chunk_list>
```

where <chunk_list> is the comma-separated list of chunks “indexes” in the SearchResult’s ChunkMap field, sorted in ascending order. Chunk indexing starts at 0.

- When two full match are detected for a search, print:

```
SEARCH FINISHED
```

- When downloading a file, print :

```
DOWNLOADING <filename> chunk <x> from <peer>
RECONSTRUCTED file sharedFile.txt
```

Output example for file search:
An output for `./client -UIPort=8080 -keywords=share,ex` could be (assume sharedFile.txt has 4 chunks, 1ex23.txt has 2 chunks and ex.txt has 7 chunks; colors are simply for helping to follow the output)

```
FOUND match sharedFile.txt at node12
MetaFile=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318 chunks=2
FOUND match ex.txt at node3
MetaFile=abc6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5123 chunks=7
DOWNLOADING MetaFile of sharedFile.txt from node12
FOUND match ex.txt at node4
MetaFile=abc6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5123
chunks=2,3
FOUND match 1ex23.txt at node5
MetaFile=6e1950ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be51234
chunks=1,2
DOWNLOADING MetaFile of 1ex23.txt from node5
DOWNLOADING MetaFile of ex.txt from node4
```

```
FOUND match sharedFile.txt at node77
MetaFile=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318
chunks=1,2,3
FOUND match sharedFile.txt at node33
MetaFile=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5318
chunks=2,4
FOUND match ex.txt at node43
MetaFile=abc6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5123
chunks=1,2
FOUND match 1ex23.txt at node77
MetaFile=6e1950ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be51234 chunks=2
SEARCH FINISHED
```

Explanation: the gossiper increased the budget up to 4, and with a budget of 4 found all chunks for 2 matched files (recall we use a threshold of 2): 1ex23.txt and sharedFile.txt. Because it already found all chunks for 2 files, it didn't increase the budget further, to find all chunks of ex.txt. Files 1ex23.txt and sharedFile.txt are thus available for download.
Note there is an end-of-line character (`'\n'`) only between the lines starting with 'FOUND', 'DOWNLOADING' and 'SEARCH'.

Output example for file download:

```
./client -UIPort=8080 -file=sharedFile.txt
-request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe699
2be5
```

```
DOWNLOADING sharedFile.txt chunk 1 from node77
DOWNLOADING sharedFile.txt chunk 2 from node12
DOWNLOADING sharedFile.txt chunk 3 from node77
DOWNLOADING sharedFile.txt chunk 4 from node33
RECONSTRUCTED file sharedFile.txt
```

Watchers

The testing framework uses the Watchers. Just as in Homework 1, you need to send all the incoming packets to `inWatcher` and all the outgoing packets from your gossiper to `outWatcher` via the `Notify` method of the watcher. Specifically, you need to wrap the gossip packet in the `CallbackPacket` struct that you will find in `gossip/packets.go`. You do not need to modify the code of the watcher, simply send in the incoming and outgoing gossip packets.

General remarks

To test your search and downloading code effectively, set up a test topology containing several nodes connected only indirectly, and make sure you can search and download across indirectly connected nodes. Put several files on each node, and make sure the filenames have both "common" and "rare" keywords as substrings. When you search for a keyword matching several files, make sure that matches at "nearby" nodes tend to appear first (as they should due to the expanding ring search), but that you eventually see all the matches (e.g., after several seconds as the search budget increases to encompass the entire test network).

Automatic testing through GitLab

We will use GitLab for testing. Each of you will be provided with a GitLab repository that you can access with your credentials for gitlab.epfl.ch. The CI (continuous integration) tool will automatically run tests every time you push your code in the repo. For your convenience in implementation, you can inspect the tests and change them to debug your program. Unit tests and integration tests will be available soon.

We will provide you with the basic code structure with interfaces, on top of which you can write your code. To facilitate smooth code merging and in order for your code to work with the testing infrastructure, ***we strongly advise you not to change the project structure***.

Hand-in Procedure and Grading

For each homework, you will receive a grade out of 6. To be considered for receiving full points (max. 6), you must upload and submit your fully-working code on Moodle (simply as a collection of source files) by the due date, which you can find at the beginning of each homework assignment document. **You can always update your submission on moodle until the deadline, so please start submitting early. Late submissions are not possible.**

We will grade your solutions via a combination of automatic testing, code inspection, and code plagiarism detection. We will run automated tests on your application and make sure it works as required, both when communicating with other instances of itself (unit tests) and when communicating with our own instances (integration tests). You will be provided with the test framework, so that you can test your code yourself before submitting. Unless otherwise specified, the tests assume a full implementation of the homework. If you implement only parts of a homework, we cannot guarantee there will be tests for that particular functionality that would give you points for it. For the actual grading, we will use the same tests but with *different* input data so only the implementations that correctly implement all the functionality will pass.

Our very first test is that your code must compile when ***go build*** is executed on your files. **No points will be given if your code does not compile.**

References

[1] Hash tree. https://en.wikipedia.org/wiki/Hash_tree
[2] SHA-256 in Golang's crypto package. https://golang.org/pkg/crypto/sha256/